Available online at www.sciencedirect.com
ScienceDirect
Computers and Mathematics with Applications 55 (2008) 1250–1256
An International Journal
computers & mathematics
with applications
www.elsevier.com/locate/camwa

An adaptive character wordlength algorithm for data compression

Hussein Al-Bahadili[a,*], Shakir M. Hussain[b]

[a] *Department of Computer Science, Amman Arab University for Graduate Studies, Amman, Jordan*
[b] *Department of Computer Science, Applied Science University, Amman, Jordan*

Received 15 May 2006; accepted 16 May 2007

Abstract

This paper presents a new and efficient data compression algorithm, namely, the adaptive character wordlength (ACW) algorithm, which can be used as complementary algorithm to statistical compression techniques. In such techniques, the characters in the source file are converted to a binary code, where the most common characters in the file have the shortest binary codes, and the least common have the longest; the binary codes are generated based on the estimated probability of the character within the file. Then, the binary coded file is compressed using 8 bits character wordlength. In this new algorithm, an optimum character wordlength, b, is calculated, where $b > 8$, so that the compression ratio is increased by a factor of $b/8$. In order to validate this algorithm, it is used as a complement algorithm to Huffman code to compress a source file having 10 characters with different probabilities, and these characters are randomly distributed within the source file. The results obtained and the factors that affect the optimum value of b are discussed, and, finally, conclusions are presented.
© 2008 Published by Elsevier Ltd

Keywords: Data compression; Coding; Huffman coding; Binary codes

1. Introduction

Data compression algorithms are designed to reduce the size of the data so that it requires less disk space for storage and less bandwidth to be transmitted on a data communication channel [1–3]. Data compression is usually obtained by substituting a shorter symbol for an original symbol in the data. The symbols may be characters, words, phrases, or any other unit that may be stored in a dictionary of symbols.

Data compression algorithms can be classified in two different ways. They may be classified as lossless or lossy, or may be classified according to a fixed or variable size of group taken from the original file and written to the compressed file [4,5]. Lossy compression can provide compression ratios of 100:1 to 200:1, depending on the type of information being compressed, while lossless compression techniques can usually achieve a 2:1 to 8:1 compression ratio. Lossy compression techniques are often tuneable in that they can turn the compression up to improve throughput, but at the cost of a loss in quality. Compression can also be turned downed to the point at which there is little loss of data, but throughput will be affected [6]. A detailed discussion of data compression techniques are found in [5,7].

* Corresponding author.
E-mail addresses: h.-al-bahadili@aau.edu.jo (H. Al-Bahadili), shakir@asu.edu.jo (S.M. Hussain).

0898-1221/$ - see front matter © 2008 Published by Elsevier Ltd
doi:10.1016/j.camwa.2007.05.014

Data communication and data storage applications have benefited greatly from data compression methodology. By reducing the size of the transmitted data, the effective bandwidth of the communications channel can be increased. The obvious advantage for data storage applications is that smaller data require less storage space. Thus, the effective storage capacity of any storage medium is increased if the data are compressed. There is, however, with current technology, another important implication of data compression on storage technology [3].

There are three major types of compression techniques: Substitution, Statistical and Dictionary based compression [8]. Substitution data compression techniques involve the swapping of repeating characters by a shorter representation, such as null suppression, Run Length Encoding (RLE) [4,9], bit mapping and half byte packing [8]. Statistical data compression techniques involve the generation of the shortest average code length based on an estimated probability of the characters, such as Shannon–Fano coding [5,10,11], static/dynamic/adaptive Huffman coding [12–15], and arithmetic coding [16,17]. Finally, dictionary data compression techniques involve the substitution of sub-strings of text by indices or pointer code, relative to a dictionary of the sub-strings, such as Lempel–Zif–Welch (LZW) data compression technique [18–20].

Many compression algorithms use a combination of different data compression techniques to improve compression ratios. For example, Fax machines use a combination of RLE and Huffman coding to achieve compression ratios of about 10:1. Since both coder and decoder use the same modification algorithm, the decoder will adjust its decoding table in the same way as the coder, and the two will remain in sync.

In this paper, we are concerned with the statistical data compression techniques that involve the generation of the shortest possible binary code length for each character based on the estimated probability of occurrence (frequency) of the characters in the data file, such as Shannon–Fano, static/dynamic/adaptive Huffman, and arithmetic coding.

All statistical data compression techniques are based on converting 8 bits, from the binary coded file into one character in the compressed file, i.e., they use an 8 bit character wordlength. In this paper, we propose an algorithm, namely the ACW algorithm, which can be used as a complementary algorithm to enhance the compression ratio of these techniques. The compression ratio, then, can be increased by a factor of $b/8$, where b is the character wordlength, and $b \geq 8$.

In Section 2, we discuss two different classification schemes that can be used to classify data compression techniques. Also, in this section, we present a brief definition of three different types compression techniques followed by a detailed description of a statistical lossless compression technique, namely, Huffman coding. Section 3 discusses the criteria for comparing data compression techniques. A detailed description of the ACW algorithm is given in Section 4. In order to validate the algorithm, a sample problem is considered in Section 5, and the results obtained and the factors that affect the value of the character wordlength are discussed. Finally, in Section 6, conclusions are drawn and recommendations for future works are presented.

2. Data compression techniques

Data compression squeezes data so that it requires less disk space for storage and less bandwidth to be transmitted on a data communication channel. In this section, we present a brief description for the different classes and types of data compression algorithms.

2.1. Classification of data compression techniques

Data compression algorithms can be classified as either lossless or lossy. Lossless methods restore the compressed data file to exactly the same form as the original data file, while lossy methods only re-generate an approximation. Lossless methods are absolutely necessary for many types of data, for example: executable codes, word processing files, tabulated numbers, etc. In comparison, lossy methods are very useful in compressing data files that represent images and other acquired signals which do not have to be kept in perfect condition for storage or transmission purposes. This distinction between lossless and lossy methods is important because lossy techniques have higher compression ratios (i.e. are more effective) than lossless methods. The higher the compression ratio, the more errors are introduced into the original data [8].

Images transmitted over the World Wide Web (WWW) are an excellent example of why data compression is important, and show the effectiveness of lossless versus lossy compression [2,21,22].

Another way of classifying compression methods is according to a fixed or variable size of the group taken from the original file and written to the compressed file. Most data compression programs operate by taking a group of data from the original file, compressing it in some way, and then writing the compressed group to the output file.

2.2. Types of data compression techniques

There are three major types of data compression technique: Substitution, Statistical, and Dictionary based compression [8]. Substitution data compression techniques involve the swapping of repeating characters by a shorter representation, such as null suppression, RLE, bit mapping and half byte packing [4,8,9]. Statistical data compression techniques involve the generation of the shortest average code length based on an estimated probability of the characters, such as Shannon–Fano coding [5,10,11], static/dynamic/adaptive Huffman coding [12–15], and arithmetic coding [16,17]. Finally, dictionary data compression techniques involve the substitution of sub-strings of text by indices or pointer code, relative to a dictionary of the sub-strings, such as the LZW data compression technique [18–20]. In each case, relative decompression algorithms were also developed.

2.3. Example of a statistical data compression technique

The adaptive character wordlength algorithm that has been proposed can be used to enhance the effectiveness of statistical data compression techniques. In this section, we give a brief description to one of the most sophisticated and efficient statistical data compression techniques, namely, Huffman coding.

2.3.1. Huffman coding

Huffman coding is a sophisticated and efficient statistical lossless data compression technique, in which the characters in a data file are converted to a binary code, where the most common characters in the file have the shortest binary codes, and the least common have the longest. In Huffman coding, for example, the text file to be compressed is read to calculate the frequencies for all the characters used in the text, including all letters, digits, and punctuation [12–15].

The first step in building a Huffman code is to order the characters from highest to lowest frequency of occurrence. The second step is to construct a binary tree structure. This begins by selecting the two least-frequent characters: logically group them together, and their frequencies are added. Then, select the two elements that have the lowest frequencies, regarding the previous combination as a single element, group them together and add their frequencies. Continue in the same way to select the two elements with the lowest frequency, group them together, and add their frequencies, until one element remains.

There is an alternate algorithm for generating these codes, known as Shannon–Fano coding. Shannon–Fano coding achieves the same results as Huffman coding, but works from the top down, instead of the bottom up. The codes that result are exactly the same as would be obtained with Huffman coding [5].

Of course, Huffman codes can be used on any type of data, such as bytes in a graphics file [22,23]. They are not very effective if the data in the file is strongly random, since the frequencies of different characters or bytes would be close to the same, but such random data files are hard to compress by any technique. Huffman coding techniques are also used in conjunction with other compression schemes, such as RLE scheme, to improve compression ratios.

Since the average frequencies of letters in every language are distinctive and well known, it is possible to devise a Huffman code table for text files written in a specific language. A receiver can use the same default or canonical Huffman codes to decompress the message.

However, actual average frequencies of individual messages will differ from that standard language average frequency to a greater or lesser degree. That means better compressions can be obtained by analyzing a specific file and building a custom Huffman code for that file. This requires that the custom table of Huffman codes be output to the file along with the compressed data stream, to allow the decoder to uncompress the data [4,5].

Another variation on this idea is known as adaptive Huffman coding. In this approach, both the coder and decoder start with a predefined Huffman table, but both keep statistics on the frequency of different characters in the compressed data stream, and modify the Huffman codes on a continuous basis by adjusting the Huffman tree as needed. Since both the coder and decoder use the same modification algorithm, the decoder will adjust its Huffman table in the same way as the coder, and the two will remain in sync [13].

3. Criteria for comparing data compression techniques

The effectiveness of a data compression technique is dependent on the type of data that is to be compressed. Therefore, to be able to compare different data compression techniques reliably, not allowing extreme cases to cloud or bias the technique unfairly, certain issues need to be considered, such as the algorithm complexity, the computational CPU time, and amount of compression [7,8]. Practically, other conditions must be considered may include: finite memory, error control, type of data and compression style (adaptive/dynamic, semi-adaptive or non-adaptive). In addition, these conditions have to be equally considered for the decompression process [24,25].

The data compression ratio is used to quantify the amount of compression achieved by a particular compression algorithm, i.e., the reduction in data quantity produced by a data compression algorithm. Data compression ratios are generally expressed as <original size>:<compressed size>. Compression ratios may be given in percents.

Data compression ratios are closely related to the concept of coding rates, which express the same concept but relate the ratio to more tangible quantities. For example, for a text file, the coding rate may be expressed in "bits per character", where typically in uncompressed text file a coding rate of 8 bits per character is used. The coding rate of an audio stream may be expressed in "bits per second", or still image compression may be expressed in "bits per pixel". In many instances, coding rates can be converted to compression ratios if the physical characteristics of the content of the original file are known. In contrast to compression ratios, lower coding rates indicate a higher level of compression.

There are extreme cases where data compression works very well or other conditions where it is inefficient; therefore, it is important to select the most appropriate technique for the correct data profile (file type). In fact, with random information it is difficult to estimate a precise value for the amount of compression [24].

The type of data that the file contains has an appreciable effect on the efficiency of the technique selected to apply to it. This is due to the practicality of the techniques, to the data, as the technique may try to act on information but may not be able to find anything suitable to compress or it may try but not very well. In the worst case, the file could be inflated in size.

The types of files that are commonly used can be classified into two main categories: binary or textual. These further can be classified with respect to their purpose and format, i.e., text, image, audio and executable. Some of these files may contain repeating sets of various characters or of the same character. In addition, repeated strings and sub-strings may also be present. In other cases, certain characters may have probabilities that are higher than others, thereby prompting the use of shorter codes for them. Each case is dependent on the context of the information contained within it; therefore, some patterns may be common across files of these types prompting a certain technique to deal with them all effectively. This is normally rare for optimum compression or speed, as there will always be a trade-off between the two, depending on the demanding nature of compression.

Investigation on the effects of each method for each of these file types can be determined by carrying out an experiment by applying each compression and decompression algorithm for each method to each of these file types. Note that an initial selection of files needs to have been made to maintain a fair comparison though the experiment. The time taken for the process to complete, and the compression ratio, are monitored and recorded. The results are then correlated with respect to time and compression for each algorithm. This will help in determining a relationship between these two factors and enable the most appropriate to be used by the user. The user will be presented this information in the application so that a decision can be made on what trade-offs to make [8].

4. The adaptive character wordlength (ACW) algorithm

In statistical compression algorithms, it is required first to find the probabilities (i.e., frequencies, f) for all characters used in the data file. For text files, characters include all letters, digits, and punctuations. These characters are to be ordered in the sequence (s) from highest frequencies to the lowest. The second step is to find the equivalent binary code for each character according to the statistical data compression technique that has been used, e.g., Huffman coding, where the most common characters in the file have the shortest binary codes, and the least common have the longest.

Then these binary codes are used to convert all characters in the data file to a binary code. Typically, in all statistical algorithms, a character wordlength of 8 bits is used, where the equivalent decimal value for each 8 bits (0–255) is calculated and converted to a character that is written to the output file.

The size of the original data file (S_{data}) in bytes is given by

$$S_{\text{data}} = \sum_{i=1}^{n} f_i. \tag{1}$$

In general, if we exclude the header of the compressed file, the size of the compressed file (S_{comp}) is given by

$$S_{\text{comp}} = \frac{\sum_{i=1}^{n} w_i f_i}{b} \tag{2}$$

where w_i is the numbers of bits used to represent each character, f_i is the character frequency, n is the number of characters within the data file, and b is the character wordlength in bits (typically 8 bits).

The compression ratio (C) is given by

$$C = \frac{S_{\text{data}}}{S_{\text{comp}}}. \tag{3}$$

Entropy is a measure of the information content of the data file and the smallest number of bits per character needed, on average, to represent the compressed file. Therefore, the entropy of a complete data file would be the sum of the individual characters' entropy. The entropy of a character is represented as the negative logarithm of its frequency and expressed using base two. Where the frequency of each character of the alphabet is constant, the entropy is calculated as [24]

$$E = -\sum_{i=1}^{n} -\rho_i \log_2(\rho_i) \tag{4}$$

where ρ_i is the probability and it is calculated as $\rho_i = f_i/S_{\text{data}}$. For an 8 bits character wordlength, there exist 0 to 255 possible decimal values, i.e., 256 possibilities. In most cases, not all the 256 possible decimal values are used. Therefore, in this new approach, we try to use more than 8 bits to store a single character, say, for example, b bits, which give 2^b possibilities (from 0 to $2^b - 1$ possible decimal values). This is not an issue; what is important is the number of the different decimal values (possibilities) that have been calculated, and if they are less than 256 possibilities, it is possible to use a character wordlength of b bits to store the character rather than 8 bits.

It is important to find an optimum value for b which produces decimal values that are less than or equal to 256 possibilities. There are different ways to find the optimum value of b, such as increasing the number of bits one by one, starting at 8, until more than 256 possibilities are found, then the previous value of b is considered as the optimum character wordlength. These decimal values that have been calculated using b bits character wordlength are arranged in descending order and stored in the header of the compressed file with other information. The compressed character that is written to the compressed file is calculated according to the order (sequence number) of that decimal value and not its actual value. Therefore, we can use more than 8 bits per character to decrease the size of the compressed file and then increase the compression ratio, as can be seen from Eqs. (1) and (3), respectively.

Fig. 1 summarizes the ACW data compression algorithm as it is used to enhance the effectiveness of Huffman coding. Steps 1–4 can be modified according to the statistical data compression algorithm used to construct the equivalent binary code.

In this new algorithm, a coder will go through the data file, convert each character into its appropriate binary code, and write the resulting bits to the output file after optimizing the compressed character wordlength. In order not to get the codes mixed up in the decoding process, all information is stored in the compressed file header. Thus, as long as bits constitute legitimate codes and a bit does not get scrambled or lost, the decoder will never get lost either.

This type of coding can be used on any type of data, such as bytes in a graphics file, video files, executable files, etc. It is also very effective if the data in the file is strongly random, since the frequencies of different characters or bytes would be close to the same, but such random data files are hard to compress by any technique.

1. Calculate the characters frequencies (f_i for $i = 1, 2, 3, \ldots, n$), where f_i is the character frequency and n is the number of the types of characters used in the original data file.
2. Order the character frequencies in sequence from the highest frequency to the lowest frequency.
3. Construct the Huffman binary tree
4. Construct the binary code for the original data file using Huffman coding.
5. Calculate the optimum value for the character wordlength, b.
6. Order the b bits decimal values that have been obtained in descending order.
7. Convert each b bits to a single character according to the order (sequence number) of its equivalent decimal value rather than its actual value.

Fig. 1. Adaptive character wordlength (ACW) algorithm.

Table 1
Character probabilities and their equivalent Huffman binary code

#	Character	Probability	Huffman binary code
1	A	0.171	00
2	B	0.031	111110
3	C	0.057	0110
4	D	0.092	1110
5	E	0.274	10
6	F	0.052	0111
7	G	0.042	11110
8	H	0.130	010
9	I	0.149	110
10	J	0.002	111111

5. Results and discussions

In order to evaluate the performance of the ACW algorithm, the algorithm is used as a complementary algorithm to a statistical data compression algorithm, namely, Huffman coding, to compress a data file with the characters probabilities given in Table 1 [5]. For an 8-bit compressed character wordlength, Huffman coding achieves a code rate of 2.94 bits per character or a compression of 2.72:1. For this data file, Eq. (3) yields an entropy lower bound of 2.88 bits per character.

The ACW algorithm will achieve a coding rate of $(2.94 \times 8)/b$ or a compression ratio of $b/2.94$, where b is the optimum character wordlength. Thus, for 9 and 10 character wordlengths, for example, the coding rates are 2.16 ($C = 3.06$) and 2.35 ($C = 3.40$), respectively. The optimum value of b depends on a number of factors:

1. The size and the type of the data file,
2. The characters frequencies within the data file,
3. The distribution of characters within the file, and
4. The equivalent binary code used for each character.

It is clear that using the ACW algorithm, the compression ratio, for any statistical compression algorithm, increases linearly with b, and it can be 4 times its original value if b is equal to 32 bits.

In order to decompress the file efficiently, the number of possibilities and their decrementally ordered actual decimal values must be stored into the header of the compressed file. The maximum added overhead occurs when the value of b is 32 bits and all the possibilities (0–255) are used. The maximum overhead is equal to 2563 B. This can be reduced to 2050 B, if these values are stored in hexadecimal notation.

6. Conclusions

In this paper, an ACW compression algorithm is presented. This algorithm can be used as a complementary algorithm to any statistical lossless data compression algorithm, such as Shanon-Fano coding, static/dynamic/adaptive Huffman coding, arithmetic coding, a combination of these algorithms, or any modified form of them, etc.

In this algorithm, after the construction of the binary code for the data file, then instead of calculating the equivalent decimal value for each 8 bits and converting it to a single character (8 bits character wordlength) to be stored in the compressed file, the algorithm searches for the optimum character wordlength, b. The value at which the binary file is divided into a number of blocks of b bits length. The equivalent decimal values for these blocks are calculated, which should be not more than 256 different decimal values (256 possibilities). These values are arranged in descending order, and then the equivalent characters to be stored in the compressed file are calculated according to the sequence number of the character, but not according to its actual value. The number of possibilities and their values are added as an overhead to the compressed file. This overhead is not more than 2 KB.

Using the ACW algorithm as a complementary algorithm to any statistical compression algorithm enhances the compression ratio by a factor of $b/8$, and decreasing its code rate by a factor of $8/b$. The value of b depends on many factors, such as:

1. The size and the type of the data file.
2. The frequencies of the characters within the data file.
3. The distribution of characters within the file, and the equivalent binary code used for each character.

However, further research studies are required to explore the effect of these factors.

References

[1] K. Sayood, Introduction to Data Compression, Morgan Kaufmann Publishers, 2000.
[2] G. Held, T.R. Marshell, Data and Image Compression, John Wiley and Sons, 1996.
[3] R. Gryder, K. Hake, Survey of Data Compression Techniques, ORNL/TM-11797, 1991.
[4] S.W. Smith, The Scientist and Engineer's Guide to Digital Signal Processing, California Technical Publishing, 1997.
[5] http://www.data-compression.com/lossless.shtml.
[6] http://www.vectorsite.net/ttdcmp1.html.
[7] D.A. Lelewer, D.S. Hirschberg, Data compression, ACM Computing Surveys 19 (3) (1987) 261–296.
[8] M.K. Pandya, Data compression: efficiency of varied compression techniques, Formal Report, Brunel University, 2000.
[9] http://ics.uci.edu/~dan/pubs/DC-Sec1.html.
[10] L. Rueda, J. Oommen, A fast and efficient nearly-optimal adaptive Fano coding scheme, Information Science 176 (12) (2006) 1656–1683.
[11] L. Rueda, J. Oommen, A nearly-optimal Fano-based coding algorithm, Information Processing and Management 40 (12) (2004) 257–268.
[12] D.A. Huffman, A method for the construction of minimum-redundancy codes, Proceeding IRE 40 (9) (1952) 1098–1101.
[13] D.E. Knuth, Dynamic Huffman coding, Journal of Algorithms 6 (2) (1985) 163–180.
[14] J.S. Vitter, Dynamic Huffman coding, Journal of ACM 15 (2) (1989) 158–167.
[15] J.S. Vitter, Design and analysis of dynamic Huffman coding, Journal of ACM 34 (4) (1987) 825–845.
[16] P.G. Howard, J.S. Vitter, Arithmetic coding for data compression, Proceedings of the IEEE 82 (6) (1994) 857–865.
[17] I.H. Witten, R.M. Neal, J.G. Cleary, Arithmetic coding for data compression, Communications of the ACM, Computing Practices 30 (6) (1987) 520–540.
[18] J. Ziv, A. Lempel, Compression of information sequences via variable-rate coding, IEEE Transactions on Information Theory 24 (5) (1978) 530–536.
[19] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory 23 (3) (1977) 337–343.
[20] M. Nelson, LZW data compression, Dr Dobb's Journal 14 (10) (1989) 62–75.
[21] G.W. Drost, N.G. Bourbakis, A hybrid system for real-time lossless image compression, Microprocessors and Microsystems 25 (1) (2001) 19–31.
[22] H. Yu-Chen, C.C. Chang, A new lossless compression scheme based on Huffman coding scheme for image compression, Signal Processing: Image Communication 16 (4) (2000) 367–372.
[23] A.Z. Asli, S. Alipour, An effective method for still image compression/decompression for transmission on PSTN lines based on modification of Huffman coding, Computer and Electrical Engineering 30 (2) (2004) 129–145.
[24] T.C. Bell, I.H. Witten, J.G. Cleary, Text Compression, Prentice Hall Adv. Ref. Series: Computer Science, 1990.
[25] T.C. Bell, I.H. Witten, J.G. Cleary, Modeling for text compression, ACM Computing Surveys 21 (4) (1989) 557–591.